Game Optimization and Steam Publishing for Swarmlake (2018)

Dominique Grieshofer*
TU WIEN

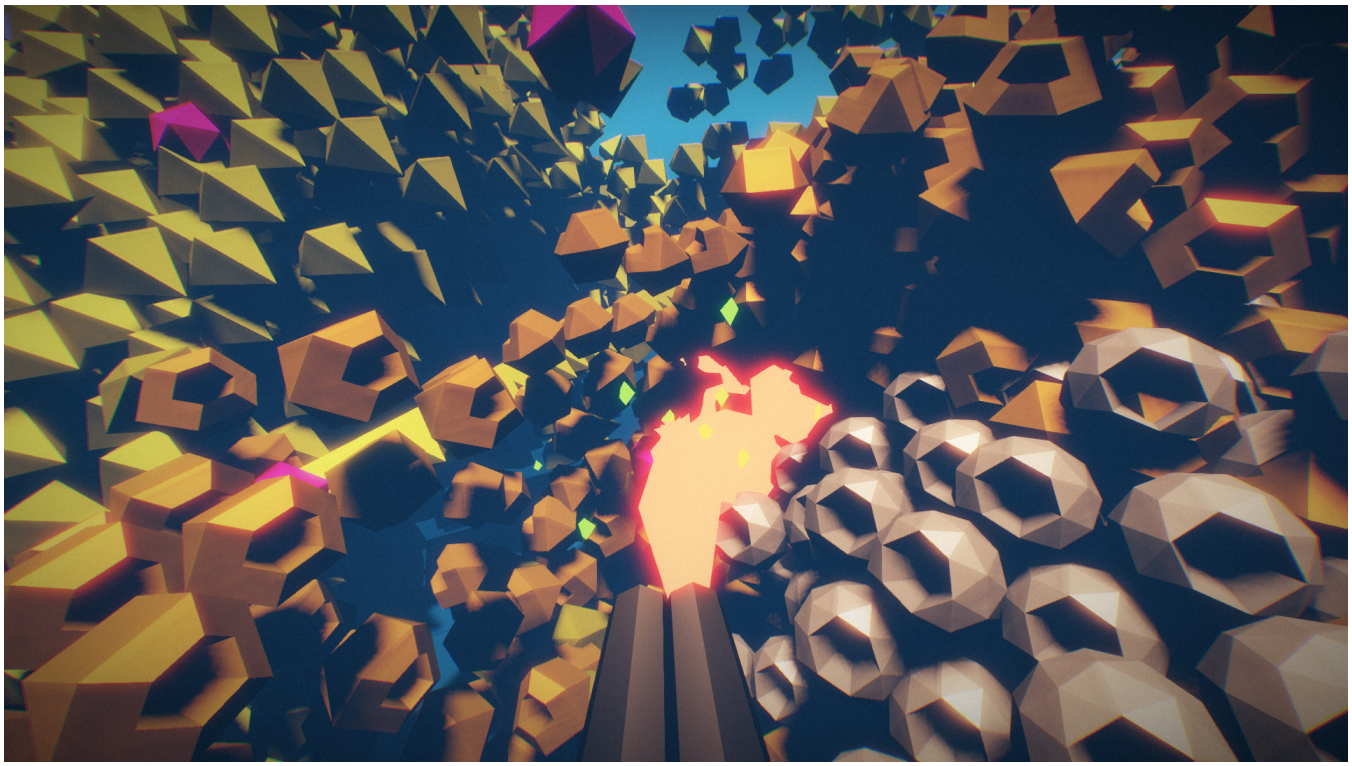

Figure 1: 10,000 enemies approaching the player in Swarmlake

Abstract

Video games are complex pieces of software which require a certain amount of prototyping and iteration to create the intended experience. They are also real-time applications and need to be performant to run at the desired speed.

Most software architecture is about creating more flexible code and therefore making fewer assumptions which allow for faster prototyping and iteration time. However, optimizing is all about making assumptions and knowing limitations to be able to improve efficiency.

Since optimal optimization is usually more natural to guarantee after making a well-designed game than vice versa, keeping the code flexible until the end is a valid compromise. Knowing game optimization patterns beforehand can be useful to make sure only the least amount of code needs to be rewritten at the end of a game's development cycle.

Successfully selling a product such as a video game also requires marketing and distribution. One of the most influential platform to distribute computer games on PC is Steam. Knowing more about the target platform a game releases on can make it more likely to make the optimal decisions in that process.

This article aims to take a look at game optimization as well as Steam integration and publishing in the context of the commercially released game Swarmlake, which is shown in figure 1.

Keywords: C++, OpenGL, CPU, GPU, game engine, optimization, Steam

1 Introduction

Swarmlake started as a university subject task of creating a 3D game from the ground up including a custom-made engine. The development team already had experience with shipping commercial games using commercial engines and were motivated to make an equally well-received game.

Without prior experience, the custom-made engine would unlikely be able to compete in aspects such as rendering, tools pipeline, and other technology. The chosen strategy to achieve the goal was to try to make a commercial game not easily possible with commercial game engines.

*e-mail: d.grieshofer@gmail.com

Since those engines need to be generic and flexible to facilitate a wide variety of different game types, they come at the cost of performance. Therefore the team concentrated on creating a game that required much optimization such as featuring more than 10,000 concurrent enemies, which later became the unique selling point of Swarmlake.

On a technical level, the engine needed to run as fast as possible with the least amount of input-delay to create the intended experience. The game also had to look modern without being too taxing on low-end hardware to be able to sell it to the most amount of potential users.

Both of these points required careful optimization on the CPU and GPU, which can be beneficial for other games as well. Since Swarmlake needs to run on as many machines as possible, it was decided to use the GPU exclusively for processing graphics, and spend most of the game calculations and optimization effort on the CPU, which is reflected in the length of that section in this article.

All results were measured on an Intel Core i5-4570 and a Nvidia GTX 1050 at the enemy engine limit of 20,000 enemies. Finally, to be able to sell the game it needed to be distributed, and Steam was chosen as the platform to do this.

Costumers on Steam expect specific functionality from their games such as having a Steam overlay, achievements and more provided through the Steamworks SDK integration. Correctly marketing a game has also never been more critical due to Steam being an almost entirely open platform by now.

Therefore research into the Steam platform was conducted, which should also be of interest to other game developers.

2 Engine Architecture

2.1 Game Loop

Video games are interactive media that come in many different shapes and designs. One software architecture almost exclusively used by almost all games is the so-called game loop.

The idea behind this pattern is to achieve a constant game time which is independent of user input and processing speed. Earlier games were sometimes explicitly programmed to work for a particular machine such as early video game consoles, meaning that the game would run at entirely different speeds on any other hardware [Nystrom 2014].

Nowadays it has become essential to make the game run consistently across a wide variety of hardware. The first adventure game, Colossal Cave Adventure, which can be seen in figure 2, was entirely based around text and waited for the player input in the game loop [Pearson 2017].

Most modern games nowadays need to keep running even without user input. Even in turn-based games, the audio needs to keep playing, and visuals such as animation need to move all the time.

From a technical point of view, the game loop consists of the following in this order:

1. An infinite loop
2. Processing player input
3. Simulating the world
4. Rendering the results on the screen

Figure 2: Colossal cave adventure [Pearson 2017]

Each loop iteration the state of the world advances based on the game time, which can be equal to real-time or scaled to real-time for effects such as slow motion.

The infinite loop is usually only terminated when closing the running game and deinitializing the application. Since a frame is usually rendered in each game loop iteration, it also means that in the context of a game, the number of frames per seconds being displayed correlates to the number of game loop cycles executed in real-time seconds.

Having an unlocked framerate means that the game loop cycles as quickly as it possibly can, which is mostly only dictated by hardware limitations. However, having an unlocked framerate and requiring the GPU to display frames as quickly as possible can lead to coil whine.

Coil whine is an unintended noise caused by an electronic component which is vibrating due to power running through an electrical cable. The component in question is usually regulating the power such as a transformer or inductor.

Coil whine is usually not audible by humans due to the specific frequency and low volume and happens in almost all electrical devices. However, high-powered components in modern PCs such as powerful graphics cards can cause these high-pitched noises to become noticeable [Crider 2017].

It is possible and also desirable to lock the framerate to a maximum intended amount to reduce coil whine and also power consumption. Locking the framerate is done by spinning in a loop at the start of the frame and essentially actively waiting to hit the desired frametime.

Another option can be to call a sleep function instead of spinning and let the operating system wake up the program just before reaching the intended time. Unfortunately, the sleep time is hard to control as the exact return time is usually not guaranteed based on the operating system's scheduling and can even cause the game to be late for the next frame.

Displaying frames on screen disregarding the monitor's refresh rate can also lead to other artifacts such as screen tearing, which is displayed in figure 3, and happens when the monitor tries to display two frames at the same time. One often used solution is to enable vertical synchronization, which tells the graphics driver to wait displaying the next frame until the previous image is fully displayed.

On Windows Vista or 7 using the Aero theme and on Windows 8 or higher, vertical synchronization is enabled by default [StudioCoast 2018].

Figure 3: Screen tearing artifact, which can be seen by the displacement in the center of the image [StudioCoast 2018]

Swarmlake runs by default in windowed mode, with position and dimensions set so that the content fully fits the display size, to opt into the automatic screen tearing prevention of Windows's DWM (Desktop Window Manager) compositor.

Due to the DWM compositor copying the window's content after drawing and composition itself happening with vertical synchronization, the game still processes simulation and input as quickly as possible without producing any screen tearing artifacts [Draxinger 2015]. This solution allows users to instantly switch between programs but also reduces performance marginally due to having to share more resources with the operating system and other applications instead of when running in exclusive fullscreen mode.

The framerate achieved by a game depends on the work required for each frame and the underlying platform the software is running on. The amount of work needed furthermore depends on the complexity and quantity of all combined tasks.

The speed of the platform is being dictated by elements such as:

- CPU speed
- CPU core count
- GPU speed
- Operating system's scheduler

In order to ensure a consistent game simulation, the passage of time is used to control the rate of gameplay with the following possible strategies:

2.1.1 Fixed Timestep

The fixed timestep is the most straightforward and theoretically ideal strategy to handle timestepping if the display refresh rate matches the game loop's delta time with vertical synchronization turned on, and the simulation takes less than the real-time of one frame.

Unfortunately, as mentioned before this cannot reasonably be ensured on the wide variety of today's hardware and would cause the game to speed up or slow down depending on the time it takes to simulate the frame.

2.1.2 Variable Timestep

To fix the issue of the fixed timestep, one valid solution would be to choose the timestep based on the elapsed real-time since the last

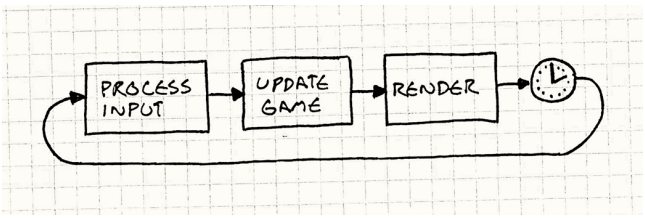

Figure 4: Variable timestep event flow [Nystrom 2014]

frame, which can be seen in figure 4. The game loop's delta time would then equal the current time subtracted by the time of the last loop's iteration.

On slow computers, this would then result in a reduced framerate while taking more significant steps in the simulation. On faster hardware, the game loop could cycle more often during the same real-time resulting in a smoother simulation with smaller iterative steps in-between.

The downside of tying the simulation to the framerate is that it becomes non-deterministic. That means that if the user performs the same inputs on subsequent plays, the game would no longer produce the same output, which can be especially problematic for multiplayer games and makes implementing replays a more complex task.

Furthermore, having extreme delta time values can cause physics to break at certain points, especially at high delta times. Rendering, on the other hand, is not affected by a variable time step since it happens after the world simulation and the displayed frame shows the current snapshot of time [Witters 2009].

For Swarmlake the variable timestep was the chosen strategy since it is the approach that results in the smallest amount of input latency. Having a non-deterministic simulation as well as no easy way to implement replays was only deemed secondary to that as it was decided early on that the former is more likely to affect the perceived fun of the game than the latter.

2.1.3 Semi-Fixed Timestep

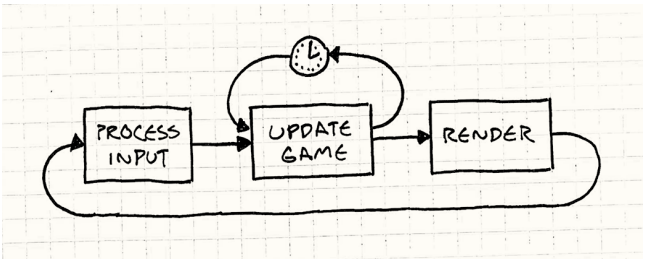

Figure 5: Semi-fixed timestep event flow [Nystrom 2014]

The semi-fixed timestep subdivides the game loop's timestep into fixed intervals which the physics simulation is known to be able to work with, followed by a final simulation step which is smaller than the fixed interval to consume the remaining time.

This is done in the update game loop, as seen in figure 5, and is a direct solution to having high delta time values by making the simulation behave well and stable in those cases.

The downside of subdividing is that it can cause the simulation to increasingly fall behind if it requires more real-time to simulate all

subdivisions than the simulated amount of game-time. Solutions to this are leaving enough headroom or clamping the maximum number of steps to a particular count, which results in slower game speed in those cases.

2.1.4 Free the Physics

The so-called free-the-physics approach decouples the simulation and the rendering of frames to achieve a deterministic simulation. Similar to the semi-fixed timestep the delta time is subdivided into fixed intervals but the remaining time is no longer processed at the end and instead added to the delta time of next game loop iteration [Fiedler 2004].

The downside of ignoring the remaining delta time is that it can cause subtle visual stuttering, as explained in figure 6, which could be fixed by interpolating between the previous and the current physics state based on the remaining time. However, this interpolation is relatively complicated to achieve since two states of the world need to be stored at all times, and it can also cause a visual latency of up to one simulated frame [Stolk 2016].

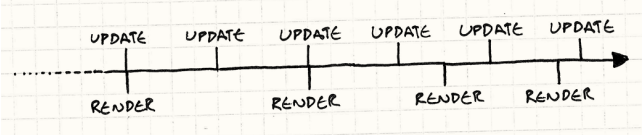

Figure 6: Free the physics game loop timeline showing the need to interpolate [Nystrom 2014]

2.2 Window/Input Handling

Making a cross-platform video game can be challenging when interacting with specific operating system implementations such as window handling and input handling. Luckily, various libraries exist which can be used to handle that burden, which includes SDL2 and GLFW.

While Swarmlake initially integrated the latter due to various bug reports on specific hardware, it became apparent that using the most stable and battle-tested library in the form of SDL2 was a better choice.

SDL2 is currently de facto standard library for low-level input and window handling, having spent 15 years in open-source development. It is officially supported by Valve, makes context management easy for GLES, Direct3D as well as OpenGL and supports operating systems such as:

- Windows
- Linux
- macOS
- Android
- iOS
- Raspberry Pi

The library abstracts away all OS events, supports relative mouse mode, which is especially useful for first-person games, uses an event loop to poll events and has a game controller API that effectively simulates all input devices as Xbox controllers, which are dominantly used by PC gamers as shown by figure 7 [Gordon 2014].

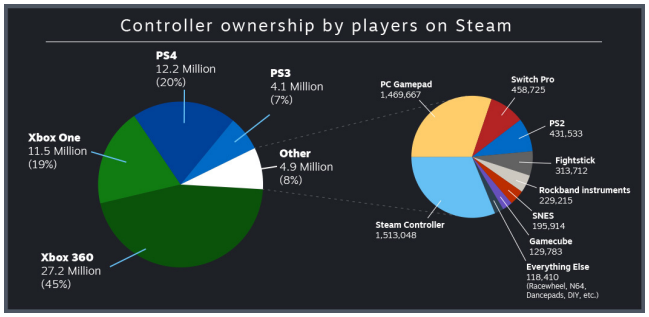

Figure 7: Controller ownership on Steam highlighting the usage of Xbox controllers on PC [Valve 2018a]

3 CPU Optimization

3.1 Data Locality

Many video games are written in C++ due to being able to handle memory explicitly and being able to write efficient and fast code if needed. The development of C++ can be summarized into the following milestones:

- 1979: Start of development
- 1983: Named C++
- 1985: First commercial release
- 1989: Release of v2.0 with additions such as multiple inheritance and abstract classes
- 1998: Standardized

Since the first inception of C++, various things have changed such as the CPU and also the memory becoming faster. However, while the speed of the former increased nearly exponentially, the latter only grew linearly, resulting in a noticeable processing and memory speed gap as shown by figure 8.

The relative memory access speed changed from about one CPU cycle in 1980 to 400+ cycles in 2009. Object-oriented programming, which is heavily used today, encapsulates code and data, which is non-ideal to get the best performance on modern hardware [Albrecht 2009].

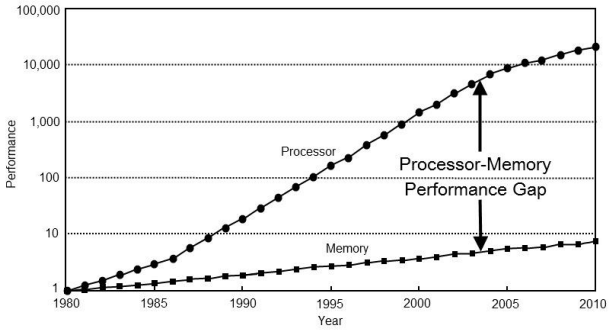

Figure 8: Historical processing speed and memory speed development [Albrecht 2009]

Data-oriented design was first coined in 2009 and is about arranging data optimally for the CPU to use caching to be able to transform data efficiently [Llopis 2009]. Whenever the CPU needs to grab

a byte, it also fetches the contiguous memory around it, which is named cache line.

If the next needed byte is in that cache line, which is a so-called cache hit, it results in faster performance than needing to fetch the next byte block from memory. The opposite result is called a cache miss and causes a CPU stall since it cannot continue processing the next instruction until receiving the needing data [Meyers 2013].

The idea of data-oriented design is that organizing data affects speed, which can be as dramatic as a 50 times performance increase in extreme cases depending on the used hardware [Nystrom 2014]. Therefore, processing should try to get contiguous data by increasing data locality.

Writing flexible code requires abstraction and in C++ that refers to interfaces or virtual method calls, which both accesses objects through pointers. Pointers involve hopping across memory, as shown in figure 9, which is called pointer chasing and can cause cache misses.

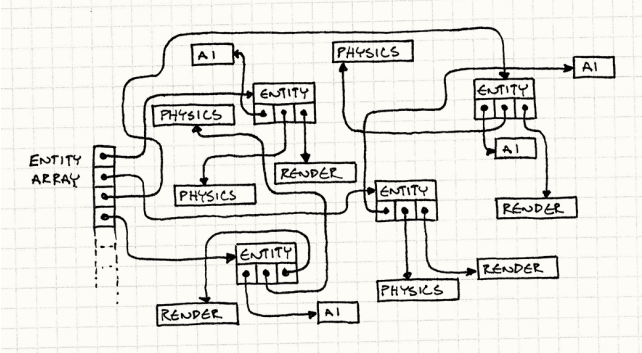

Figure 9: Pointer chasing [Nystrom 2014]

Getting rid of indirection operators (->) is usually an excellent indicator to reduce pointer chasing. Much like in other optimization patterns, improving cache locality means sacrificing flexible code abstractions, which should be carefully evaluated before making this tradeoff.

As mentioned before, having an array of game object pointers can cause cache misses since the performance depends on the memory layout. Furthermore, the memory usually gets increasingly fragmented due to frequent allocation and freeing of game objects needed in most games.

Improving data locality usually includes moving game object data, as shown in figure 10, into contiguous arrays, like displayed in figure 11, to then be able to traverse these game objects each frame quickly. This does not mean that the traditional game object entities need to be removed as they can still exist pointing into those arrays but they would need to be updated whenever game objects are added or removed.

Another option would be to have handles to game objects which are then looked up in an index table to find the correct game object. Additionally, the array of game objects should be sorted by their active state to get rid of branches and prevent branch prediction fails from happening.

This is done by removing game objects that become inactive and adding game objects when becoming active. When adding a simple counter variable to track the size of the active elements, the container could also double as an object pool where all inactive objects are simply located after the last active one.

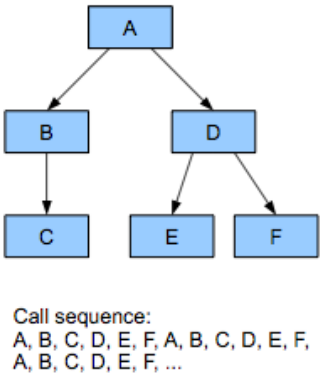

Figure 10: Object-oriented call sequence [Llopis 2009]

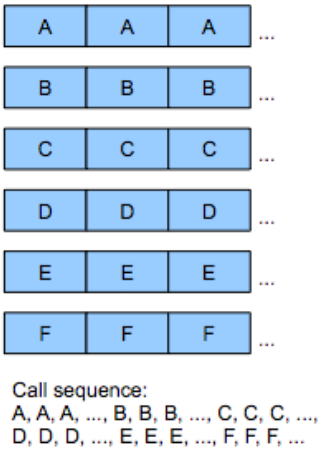

Figure 11: Data-oriented call sequence [Llopis 2009]

It should be noted that the process of removing an element of a vector with N elements can result in $O(N)$ runtime complexity due to having to move all data behind it which can be sped up. That is done by swapping with the last active one and using pop back results in a constant $O(1)$.

The side effect of using this approach is that it changes the order of the objects due to swapping, which fortunately is usually not an issue for games. When optimizing for cache locality, the amount or size of data should also be taken into account.

Having smaller data means that more information can fit into one cache line which further improves performance. In some cases, it can be useful to fold multiple variables into a single one such as using a bitmask instead of multiple bool values.

Additionally, the data could also be split into multiple arrays if part of it is only needed rarely, such as information that is only needed on construction, deconstruction or only at longer time intervals instead of at every frame. In real-world scenarios, data splitting should be done carefully since it can result in time-intensive optimization without tangible gains.

As mentioned before, specific code architecture which increases flexibility can be problematic with a data-oriented design such as polymorphism. Solutions to this include merely avoiding subclass-

ing and making separate arrays for each game object type, which has the further advantage of not needing to rely on dynamic dispatch improving the performance slightly more.

Data-oriented design usually means that processing becomes a global scope instead of a local one, on the other hand, it also results in often simpler code. Additionally, it is also more natural to parallelize that code by merely working on equally sized chunks of the data arrays in each thread, which also avoids the costly CPU cache synchronization of multiple threads requesting bytes in the same cache line [Sharp 1980].

For Swarmlake, rewriting the engine using a data-oriented design approach was tested multiple times but early results indicated an equal or lower performance, because much optimization had already been done using an object-oriented pattern. While isolated tests of iterating over 100,000 objects resulted in an approximate 100% speedup, they were in the order of microseconds, leading to believe that this is not a performance bottleneck in Swarmlake.

3.2 Object Pooling

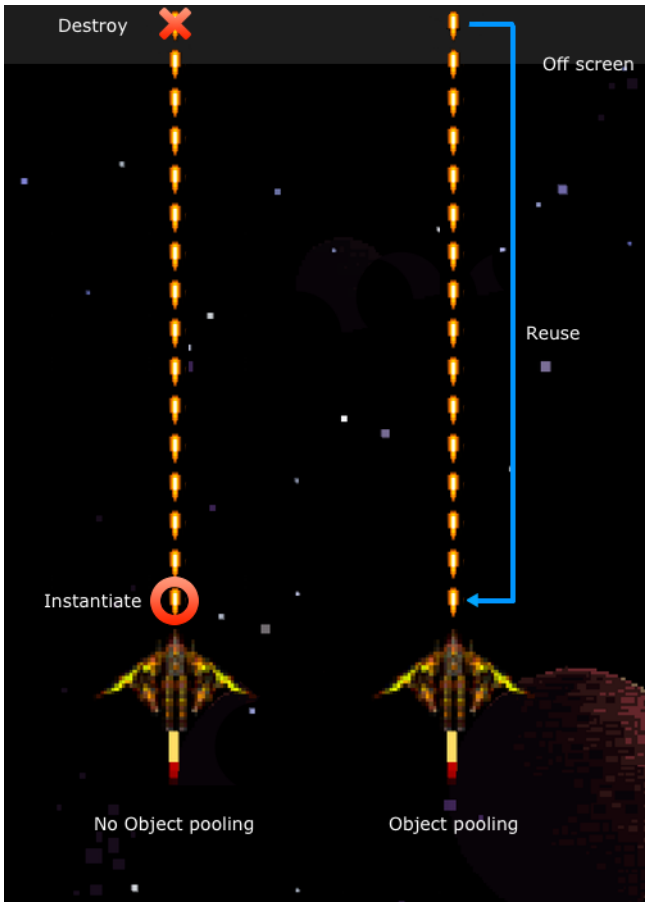

Figure 12: Object pooling comparison [Placzek 2016]

The idea of object pooling is to reuse objects from a so-called pool instead of individually allocating or deallocating objects when needed or no longer needed to improve performance, as is explained in figure 12. By falling back to the already allocated data, memory fragmentation can also be avoided.

Fragmentation causes the free memory to be scattered across RAM, reducing the availability of contiguous regions needed for more significant allocations. Reducing or avoiding this from happening further improves cache locality.

Performance gains can especially be noticed on memory-limited hardware such as:

- Embedded systems
- Video game consoles
- Mobile phones

Typically, each object pool contains a collection of reusable objects, and one game object pool is used for each type of game object.

By reusing objects, some care needs to be taken to clear or fully reinitialize them since they can still contain state from the last time they were active. When resorting to doing this manually, it is easy to forget clearing one variable on reuse, which was also the cause of minor bugs caught during the development of Swarmlake.

When the game or level is being initialized, the pool can be filled with objects which are later retrieved as needed. Complementary to that, if an object becomes inactive it can be added into the object pool again to be able to reuse it.

Adding and removing objects from the pool can also be replaced by using a state to flag if the object is in use depending on the specific needs. Various strategies deal with having all objects in use which are the following:

3.2.1 Avoid running out of objects

This strategy is about pre-allocating enough elements in the pool so that it never runs out of needed objects. For essential gameplay elements like enemies or items, this is a valid and often used approach. Ideally, the pool should be as small as possible to reduce memory and improve data locality.

The drawback is that it could result in requiring to reserve much memory depending on how varied the gameplay elements are in the game. Therefore it might be useful not to use a fixed pool size for all game objects and tune them based on the specific levels or scenarios.

For Swarmlake this was the chosen strategy for gameplay elements such as enemies and projectiles.

3.2.2 Prevent creating new objects

Not creating new objects when the pool runs out and doing nothing can be useful to avoid framespikes by preventing having too many objects active at once. For example, when a big explosion happens in Swarmlake, the concurrent audio and visual effect amounts are limited by the pool to level performance.

3.2.3 Remove less important objects

While more complicated to implement and taxing to run than the other strategies, removing less critical objects is useful mainly for sound effects where it would be more noticeable if a new sound does not play than the existing quietest one stops playing.

Additionally, the audio mix also benefits from more clarity if fewer sounds are playing at once, which was the chosen solution for the sound effects in Swarmlake.

3.2.4 Increase pool size

While causing unwanted memory allocations increasing, the pool size might sometimes be necessary depending on the type of game,

for example when featuring a big and varied open world. It should be noted that the pool could later also be reduced in size again if those additional objects are no longer needed.

3.3 Branch Prediction

On a modern CPU, a single instruction can take several clock cycles to be executed, therefore instructions are pipelined to keep the CPU busy and work more efficiently. If possible, the next instruction will start before the previous one finishes.

This means that guessing the next executed instructions is necessary, which is relatively simple for straight-line code. For control flow, guessing becomes more difficult and is based on the previous check results.

Whenever a branch prediction fails, the pipeline needs to be flushed, and the instruction will have to start over. The performance impact of this varies, but avoiding flow control in hot code can be faster.

In Swarmlake branches within the update loop about state changes were replaced with variables if possible. For example, instead of increasing or decreasing the size of an object based on a state, we simply store the intended positive or negative increment in a variable when the state changes and apply it to the current size each frame.

Additionally, timers were implemented for time-based states, which are then calling events once instead of checking a condition each frame. Sorting timers by their time also allows checking the next upcoming ones globally, which further improves performance.

3.4 C++ STL Containers

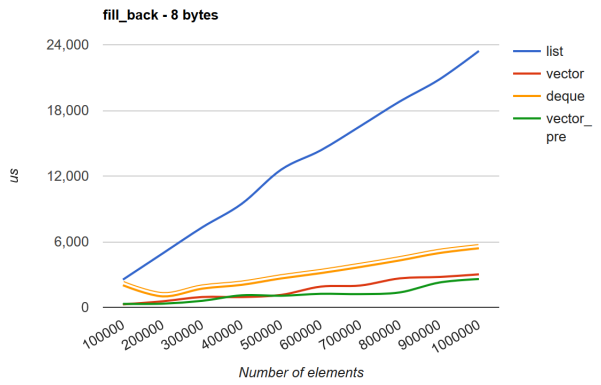

Figure 13: List, vector, deque and pre-initialized vector fill back performance [Wicht 2012]

C++ allows programmers to use the STL (Standard Template Library) to have access to certain generic containers such as:

- std::vector
- std::list
- std::deque

A C++ vector is a contiguous dynamic array which can increase in size if needed. Since reallocating more space and moving the data can be expensive, it can be faster to reserve the maximum needed capacity beforehand [Pozo 1997].

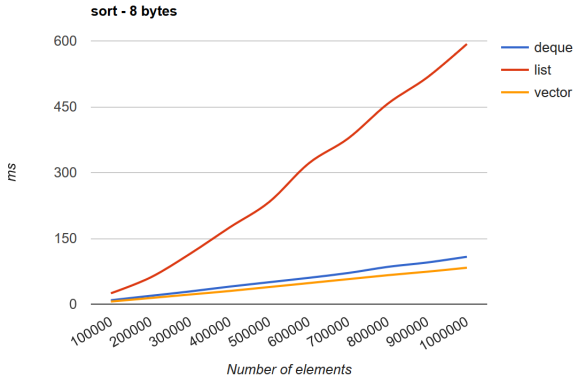

Figure 14: Deque, list and vector sort performance [Wicht 2012]

When iterating over many objects each frame a vector, an array or similar implementations are usually the most performant solution due to the contiguous memory they provide.

As mentioned before, the downside of removing elements in the middle can be avoided by using the swap and pop idiom, making them ideal for games unlike the deque or linked list as highlighted by figure 13 and figure 14, which is also why they were used for all collections in Swarmlake.

3.5 Multi-Threaded Engine

Since today's CPUs provide multiple cores to improve the maximum possible processing speed, it is required to implement a multi-threaded engine to make use of all resources.

If possible mutexes, locks and other synchronization mechanisms should be avoided to prevent possible required wait time. There exist two different strategies:

3.5.1 Functional Decomposition

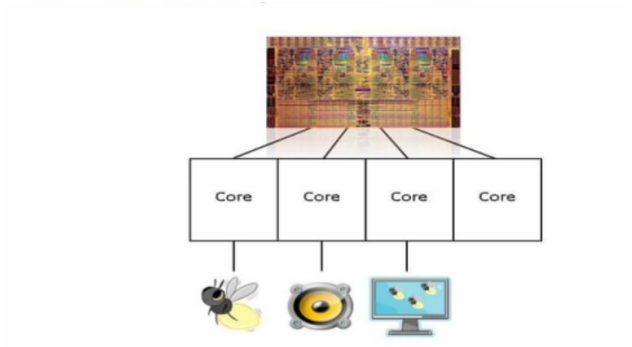

Figure 15: Function decomposition graph [Orion Granatir 2010]

Functional decomposition, as seen in figure 15, involves splitting the work by function such as AI, physics, audio, and renderer. Unfortunately, this does not scale well and would not be future proof if CPU core count increases.

3.5.2 Data Decomposition

Instead, parallelizing the data, as in figure 16, is a more effective solution, which is done by iterating over all objects in parallel while only allowing to mutate its local state but not the global state which can only be read from. Afterwards, there can be another loop executed on a single thread which then allows the objects to mutate the global state if necessary [Orion Granatir 2010].

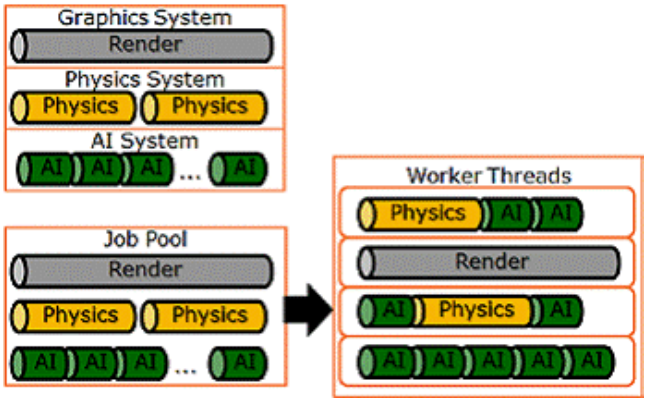

Figure 16: Data decomposition visualization [Binstock 2011]

The underlying implementation usually creates jobs to update game objects which are being executed in a thread pool. This pool spawns a thread for each physical or logical CPU core which waits for new jobs to process [Akhter and Roberts 2006].

In Swarmlake there is one thread spawned for each logical CPU core which works on jobs that process equally sized game object chunks based on the thread count. These jobs consist of simulating as much data of the objects as possible without changing global state, which includes calculating collisions, position, rotation and the render matrix.

Afterwards, all game objects are processed sequentially in the main thread to update global state such as updating the new positions in the spatial grid and starting timers if necessary. Then, the render matrices are collected in a separate array and sorted, which is done in a separate job for each game object batch to prepare rendering the frame.

Finally, the commands are sent to the GPU in the main thread to have the least amount of visual latency.

3.6 Spatial Partitioning

Many modern games need to use collision so that the player can interact with the environment. Checking N objects for a collision by the player is a task with $O(N)$ runtime complexity, and if objects themselves need to collide with all other objects as well, this increases to $O(N^2)$ when using a brute-force approach.

This expensive task can be improved by using spatial partitioning, which is about organizing and efficiently retrieving objects by their spatial location. With this paradigm, only nearby objects are checked for their collision, which speeds up the process.

Spatial partitioning is more effective the more concurrent objects are required due to the exponential nature described above. Other game systems such as audio can also benefit from having fast access to spatial information.

The active game object list used by most games is usually not sorted by their distance, which means that another container needs to be added to store the information. In that case, this performance strategy is - similar to others - a tradeoff between a bigger required memory size for more speed.

Another idea would be to only store the objects in the partitions instead of an active game object list, but having the requirement of iterating over empty partitions would reduce the data locality and might therefore not be ideal. It should be noted that the spatial data also needs to be reorganized whenever objects within it change location [Lefebvre and Hoppe 2006].

For maximum efficiency, the partitions should be balanced by having approximately the same amount of objects within them. The following strategies exist to accomplish this:

3.6.1 Hierarchical Partitioning

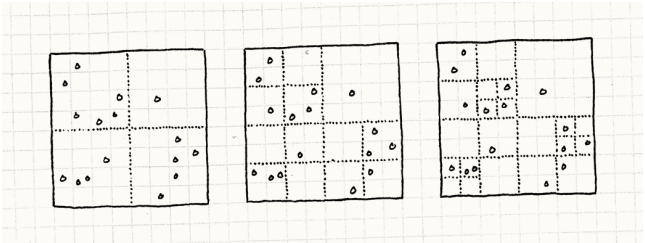

Figure 17: Hierarchical partitioning using a quadtree [Nystrom 2014]

As mentioned before, it is more efficient to avoid empty space, which is done by hierarchical partitioning implementations [Eitz and Lixu 2007]. Examples include a k-d tree or Quadtree, which is visualized in figure 17.

3.6.2 Flat Partitioning

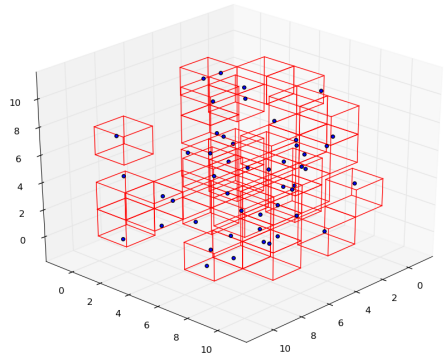

Figure 18: Spatial partitioning in 3D using a fixed grid [Jr 2016]

A flat spatial partitioning is both more straightforward to implement and understand than a hierarchical one and examples include a fixed grid, as displayed in figure 18. Additionally, the memory usage is constant since the partitions can be fixed at the start of the game or level.

Updating the positions of each object can be more performant as well since there are not multiple layers that could be affected. For

Swarmlake, this strategy was used since the game level is small and densely packed with objects having little empty space.

Collision and enemy avoidance is the most expensive system in the game, which requires about 4.4ms (68%) of approximately 6.5ms frame-time and as such, tweaking the grid dimensions, the bucket dimensions and the bucket array size allowed for significant improvements.

3.7 C++ Low-Level Optimization

Aside from the previously mentioned more significant optimization methods, there are many more small tweaks possible to speed up computation some more. Since games usually operate with a known range of possible player interaction, many calculations can be precomputed by using static tables or arrays.

The compiler can also help improve performance, for example, trying to inline smaller functions to prevent unnecessary function calls. Similarly, trivializing structs by removing manual constructors and destructors [Fernandes 2011] can allow the compiler to optimize the code even more, for example by reducing the runtime complexity of clearing a vector from $O(N)$ to $O(1)$ [Cato 2013].

Additionally, using the "fast floating-point operations" compiler optimization can help speed up mathematical equations at the cost of precision, which is also the default compiler option used by Xbox games [Hogg 2015]. In the case of Swarmlake, this resulted in about a 1ms frame-time speedup without any visual noticeable differences.

Using constant references for complex data objects such as strings when only reading and not assigning data can save on copy instructions [Chopanza 2014]. Using global variables instead of a singleton or references is also sometimes done for efficiency at the cost of code maintainability.

Using a math library that can call SIMD instructions which are used by most CPUs used today such as SSE can be beneficial [Tian et al. 2012]. More manual math simplifications can be to not calculating the square root where applicable, for example by comparing squared distances instead of actual distances.

More game-specific optimization can also be to avoid work as much as possible, such as not calculating inverse matrices for correct shadows for non-uniformly scaled objects by merely only using uniformly scaled ones. Likewise reducing excessive state checks at the cost of code maintainability might be possible, but this could also evolve into a significant time investment for little to no gain.

Exception handling and RAII (Resource Acquisition is Initialization) can be disabled in the compiler as it is usually unnecessary for games. Finally, if using a thread pool with jobs implemented as std::function, it might be useful to reduce their allocations by using template parameters, wrapping them in a lambda, using stack allocation or using std::ref() and std::cref() [Wolfe 2015].

4 GPU Optimization

4.1 View Frustum Culling

To be able to render a frame, usually a virtual camera is used which visualizes the world. With a perspective projection, the position of the camera represents the tip of the visible pyramid, which is further truncated by the near and far clipping planes, as seen in figure 19.

Those planes are required due to needing to represent depth within a finite amount of values in the depth buffer. The goal of view frustum culling is to identify all objects that are wholly or partially inside the view frustum and cull away everything else, as displayed in figure 20.

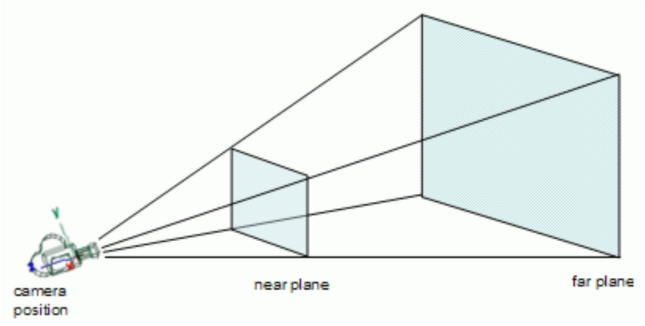

Figure 19: View frustum visualization of a camera [Lighthouse3d 2011]

Due to how the visibility is being determined in the described method, the occlusion of objects is not taken into account. That means that objects within the view could still be rendered even when being wholly occluded by other visible objects in the front.

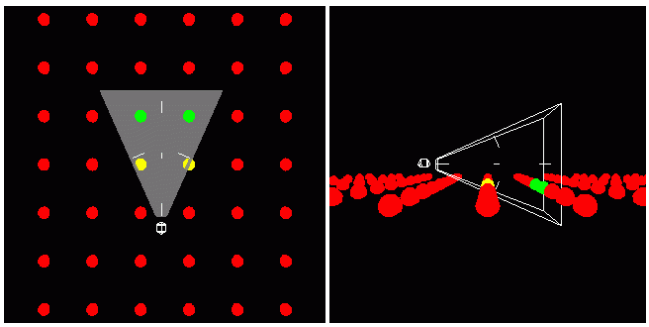

Figure 20: Top and side view of view-frustum culling [Lighthouse3d 2011]

Since some time is required to calculate the visibility of all objects, this optimization should only be used when the visible part of the world is a small enough fraction of the whole world, that the performance gains outweigh the costs [Akenine-Moller et al. 2008].

At first, the frustum volume information is being calculated whenever the camera changes, which equals to every frame in many games. Afterwards, each object in the world is tested against this updated view frustum volume.

The geometric frustum culling approach extracts the six planes of the view frustum volume boundary. The normal of each plane points towards the inside to be able to calculate the point distance using a dot product. To test if a point is inside the view the distance needs to be positive for all planes.

Doing this check is expensive for each vertex. Therefore, it is suggested to instead check with bounding volumes such as spheres or boxes. To test for spheres instead of points the distance needs to be bigger or equal the bounding volume radius for all planes [Coorg and Teller 1997].

In Swarmlake view frustum culling was implemented using this approach with bounding spheres due to its simplicity. Additionally, the near and far planes are not calculated and checked since the former does not clip away significantly more after the other sides are checked and the latter not being needed as all objects in this

specific game are within the far clipping plane of the camera. This optimization reduced the total render-time by about 1.2ms in average cases.

4.2 Instanced Rendering

To be able to render objects on the screen, the CPU needs to pass data to the GPU using API calls. Sending the data for drawing an object is called a draw call, which is a resource-intensive task that causes additional performance overhead on the CPU and can cause a bottleneck, as can be seen in figure 21.

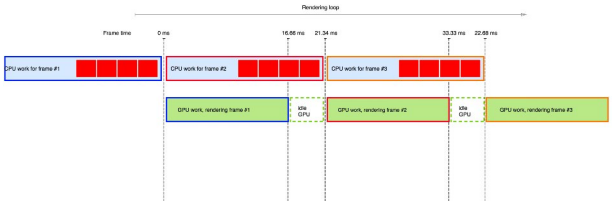

Figure 21: Visualization of a CPU that bottlenecks the GPU [Jukić 2015]

Instanced rendering uses a single draw call for all instances of the same model instead of dispatching multiple expensive draw calls while changing uniform variables for each object, as is visualized in figure 22. A single occurrence of a model within this drawn batch is a called an instance.

All of those instances use the same vertex data but have different world transformations, causing instanced rendering to be more efficient. In OpenGL, this is implemented by storing the instanced array vertex attribute in a vertex buffer object and configuring the attribute pointer to use this attribute in the shader [Wright Jr et al. 2010].

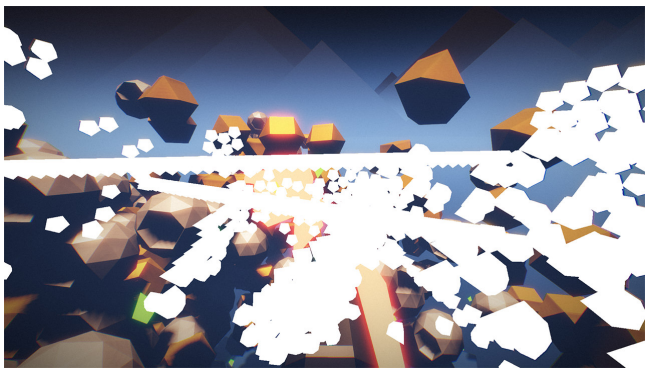

Figure 22: A single draw call highlighted in white using instanced rendering in Swarmlake

Additionally, the vertex attribute divisor can tell the GPU to automatically update the vertex attribute for each instance itself, and the render call is then being issued using separate instanced versions of the draw function [Shreiner et al. 2009]. The downside of this method is that all objects need to be static models and that they can only be individually animated in the shader [Fan et al. 2015].

In Swarmlake, instanced rendering was used for all objects that have multiple occurrences in the world, which made it possible to render ten thousands of enemies, thousands of collectibles and hundreds of projectiles each frame with little overhead. However, for performance reasons, the vertex count of all models also needed to be as low as possible due to the large number of objects, as seen in figure 23.

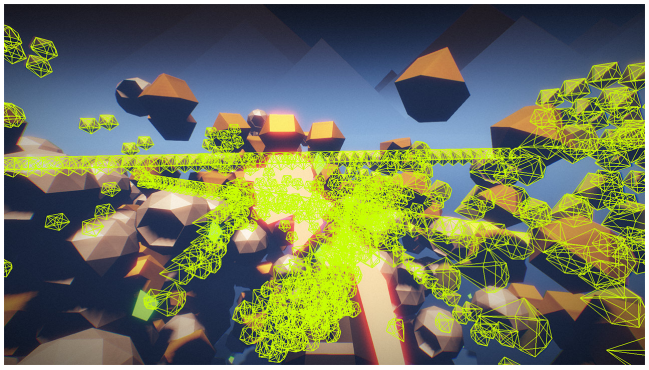

Figure 23: Wireframe of a single draw call in Swarmlake showing the small number of polygons and vertices

Finally, performance can be further improved by reducing GPU commands such as unnecessary state changes and queries for example by caching OpenGL states manually instead of requesting the current value.

4.3 Depth Sorting

Most of today's GPUs support early depth testing, which allows the hardware to run depth tests before the fragment shader is executed. If it is clear that a pixel is behind an already written, one this fragment will be discarded, which speeds up the process [Khronos 2018a].

The idea of a front-to-back renderer is to draw objects in the front first to reduce overdraw. This can be implemented by sorting all models or model instances that need to be rendered ascending by their distance to the camera before sending them to the GPU, as seen in figure 24.

Additionally, overdraw can be further reduced by using less transparent and more opaque shaders, which is especially important on systems such as consoles and mobile phones. In Swarmlake, all batches are ordered by their expected distance for example by rendering the gun first and the skybox last.

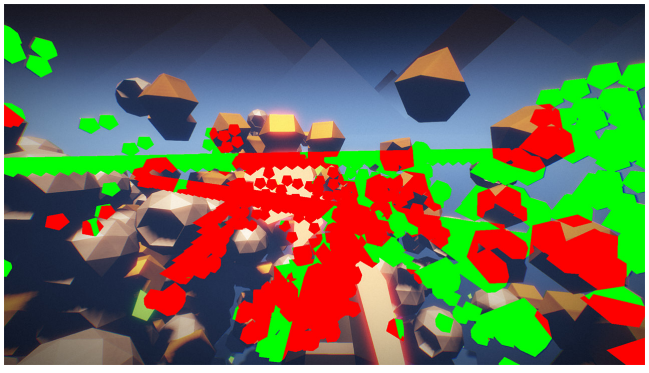

Figure 24: Early depth test result of a draw call in Swarmlake with red being discarded pixels

In case of enemies, where this cannot be predicted, sorting the batches by the used model vertex count resulted in a frame-time performance speedup of about 0.7ms. Additionally, instances within

those batches such as each enemy or projectile are also sorted by their calculated distance to the camera and all used shaders are completely opaque.

This allowed early depth testing to be effective, which resulted in a noticeable performance speedup. Finally, relying on the depth buffer can allow to not having to clear the color buffer, which further increases the render speed slightly [Paul 1997].

4.4 Bloom Downsampling

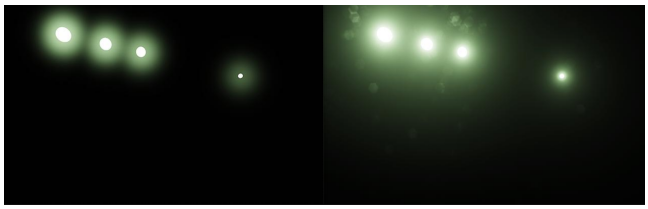

Figure 25: Bloom with a single Gaussian filter on the left compared to bloom with five Gaussian filter iterations and combined with a dirt texture on the right [Mittring 2012]

Many modern games use a subtle and wide bloom in their post process pipeline, as shown in figure 25, which has become an expected effect by players today. Implementing this with a single Gaussian blur filter can be expensive since the kernel size would need to be relatively large.

A more elegant solution is to approximate a big kernel size, as seen in figure 26, by doing the following:

1. Downsample image by 50%
2. Extract bright parts with a high-pass filter based on the luminance
3. Blur using a horizontal-pass followed by a vertical-pass Gaussian blur filter
4. Repeat multiple times in a loop

Finally, all downsampled images are additively blended on the source image to get the final look. In Swarmlake this technique reduced the total render-time by about 2ms. Further optimization includes reducing the initial bloom source image resolution before downsampling.

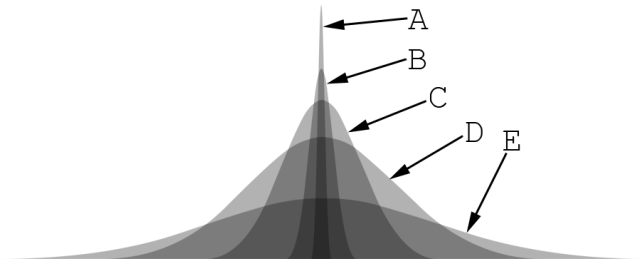

Figure 26: Bloom distribution when being downsampled five times [Mittring 2012]

Another option would be to remove the high-pass filter by squaring and then normalizing the color result in the shader that additively blends the results together. The downside of this is that it can change the bloom colors from the source color, which results in a more stylized look.

In Swarmlake, the initial bloom image is 25% the size of the source image which is being downsampled to 12.5% and 6.25% in the respective iteration of a total of three loops without using a high-pass filter.

4.5 Shader Low-Level Optimization

Shaders are being compiled by the graphics drivers before sending the data to the GPU. While the compiler can perform certain speed improvements, it cannot change operation semantics and therefore cannot replace manual optimization.

Low-level shader optimization requires deep GPU understanding and may highly depend on the hardware architecture. In some cases using a single MAD instruction instead of an ADD and MUL instruction by changing the formula can result in a speedup. This optimization cannot always be done by the compiler as it can result in unsafe code [Persson 2013].

Negations and ABS instructions are also usually free on input and SAT instruction free on output. That means that in the case of GLSL which is the shading language used by OpenGL it can be beneficial to use a clamp() function with 0.0 as minimum and 1.0 as maximum value instead of either a maximum() or minimum() function with equivalent barriers.

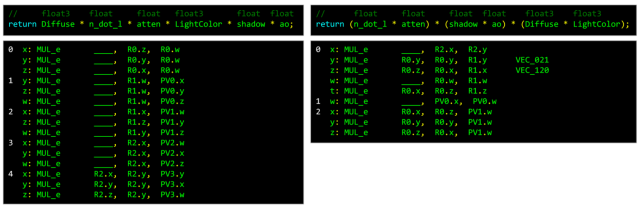

Figure 27: Shader evaluation-order optimization using brackets [Persson 2013]

Scalar and vector operations can be separated as well, as shown in figure 27, for example by using brackets to break up multiplication dependency chains in order to improve performance. Additionally, functions such as rcp(), rsqrt() and sqrt() which map to hardware can be faster than for example using normalize() [Persson 2014].

In some cases using constants where possible, unrolling loops manually and getting rid of branches using functions such as step() can also result in a speedup [Khronos 2014].

5 Steam

5.1 History

Valve was founded in 1998 by Gabe Newell, and Mike Harrington and initially made best-selling game franchises such as Half-Life, Counter-Strike, Left 4 Dead, Team Fortress and Portal. However, the company faced a problematic business model for PC games such as video game piracy, the console gaming competition, and selling games physically without the ability to update and add services or extra content [Marshall-Nagy 2014].

Therefore, as a possible solution Steam was launched in 2003 by Valve as a digital distribution platform to buy the company's games and have online gaming functionality. Additionally, it came with authentication and unobtrusive DRM (Digital Rights Management) functionality, which removed the ability to trade used games [Dunn 2013].

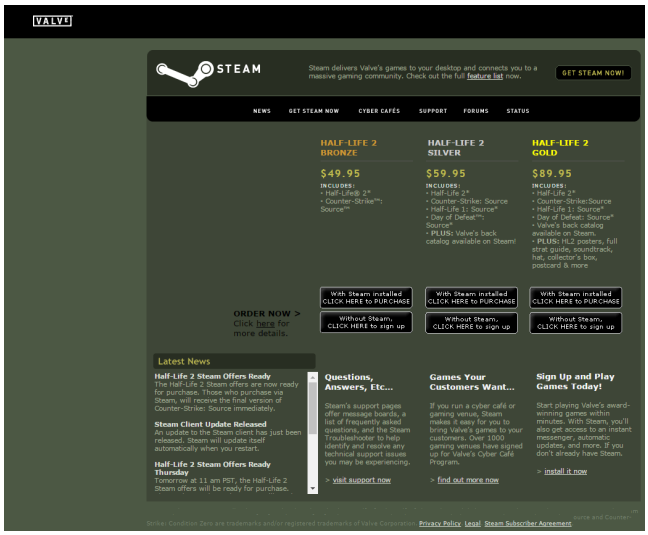

Figure 28: Half-Life 2 launch on Steam [Sayer 2016]

That was a significant risk for Valve since users were expected to dislike having less freedom with their bought software. Therefore, the company's highly anticipated game Half-Life 2 was launched exclusively on Steam to help make the platform more widely used, as shown in figure 28 [Wingfield 2012].

Figure 29: First third-party game Ragdoll Kung Fu distributed on Steam [Sayer 2016]

In 2005 games from other developers and publishers such as Ragdoll Kung Fu were added onto Steam, as seen in figure 29, which marked the transition to an online store for all PC games. Continuously more features were added such as:

- Cloud saves
- Achievements
- Gamer profiles
- Community groups
- Support for macOS and Linux

Valve is especially active in supporting Linux due to the introduction of the Windows App Store starting in Windows 8, which could be seen as a threat to the company.

Their effort includes initiatives such as building Linux Steam machines [Wilde 2018], adding support for shader pre-caching for OpenGL- and Vulkan-based games [Valve 2017d] and adding support to play Windows-exclusive games on Linux using a modification of Wine called Proton [Valve 2018b].

Due to being the first to market and Valve's continued work, Steam has cemented itself at the forefront of online gaming distribution even with an increasing amount of competitors such as EA's Origin or GOG.

5.2 Greenlight

Steam is a global digital distribution platform for PC games featuring many third-party developers and publishers. In the past, a small team at Valve decided which games would be allowed onto this store.

As the community interest increased in requesting more titles that the team would not have necessarily picked they became unsure about their own judgment and introduced a new way for games to be added.

Steam Greenlight was introduced on August 30, 2012, so that the community could help decide if a game should be published on Steam, as displayed in figure 30. There, developers or publishers would post information and media about their product and users could rate with a thumbs up or thumbs down depending on their interest [Valve 2012b].

Figure 30: Steam Greenlight Portal [DellaFave 2014]

These potential buyers were able to leave feedback in the form of comments and forum discussions. That allowed developers and publishers also to gain exposure and connect with their possible future customers.

Steam Greenlight substantially lowered the barrier for developers to release games on Steam. On the first day alone more than 600 games were submitted with a total of over 2.3M user votes.

Therefore, one week after introducing this new process a one-time submission fee of 100$ was added due to concerns about discoverability and fraudulent or gag listings. This fee entirely went to the

charitable organization Child's Play and needed to be paid by the developer or publisher for each product [Valve 2012c].

On September 11, 2012, the first ten games successfully made it through Greenlight based on the community votes and Valve's final decisions. Out of those, the first publicly released game was McPixel on September 26, 2012 [Valve 2012a].

To create a product listing on Steam Greenlight, as shown in figure 31, the following was needed:

- A valid Steam account
- Filling out the submission form
- A box art image
- One Youtube video and at least four screenshots
- A description of the product

Figure 31: Steam Greenlight page of a successfully accepted game [DellaFave 2014]

This listing needed to rank under the top currently votable games in order to be looked at by Valve, who made the final decision. With about 1,500 games being eligible at any time in 2013 Valve usually accepted batches of 100 games or products, per month.

The top 50 of those games had a 60:40 ratio of thumbs up to thumbs down votes, but only thumbs up votes were counted towards the rank. Therefore driving traffic and attention towards the Steam Greenlight page using the developer's social following or Kickstarter campaign was suggested [DellaFave 2014].

The graphical and overall presentation of the Steam Greenlight page such as the trailer, a clear USP (unique selling point) and a concise elevator pitch was important since users could not try the proposed games.

Steam Greenlight was active for five years during which more than 90M votes were recorded by at least 10M players. 63M players played games released through this process and over 100 of those titles made over $1M each [Valve 2017b].

Many of the games released through Steam Greenlight were unlikely to be published on the earlier heavily curated store.

5.3 Steam Direct

To make the process of getting a game on Steam as accessible and transparent as possible Valve introduced Steam Direct on June 13, 2017, which replaced Steam Greenlight [Valve 2018g]. The idea behind this system is that neither the platform's company nor the community has a say in which types of games are allowed onto the store.

While Valve distances itself from the created and sold content in general, everything is allowed onto the store except for illegal or trolling content [Valve 2018n] such as:

- Incorrectly labeled or age-gated adult content
- Defamatory statements
- Infringing content
- Content showing child exploitation
- Malware
- Viruses
- Fraudulent content

The requirements of Steam Direct include digital paperwork such as a company name and address as well as contact, bank, and tax information. Similar to Steam Greenlight a $100 fee per application is still required but the payee will be able to recoup it after at least $1,000 revenue [Valve 2018c].

After getting access to the Steamworks page, the developer can upload the build of the game and set up the store page. This submission is then being reviewed to check if the configuration is correct and to make sure that it does not include illegal content.

Additionally, the store page of the game needs to be visible for at least two weeks before release to make it possible for the community to report possible issues. Finally, there is also a 30-day wait duration between signing up for Steam Direct and being able to release the title.

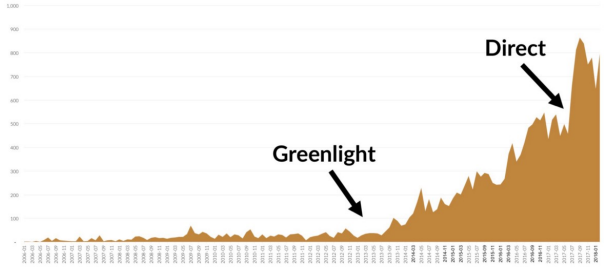

Figure 32: New games on Steam being released each month [Galyonkin 2018]

Steam Direct resulted in more games than ever released on Steam, with 7,696 in 2017, which is 39% of all of the available 21,406 games, as displayed in figure 32 [Galyonkin 2018]. With this new process, about 180 games per week are released, which is a noticeable increase of the approximately 70 games per week during Steam Greenlight, as shown in figure 33.

However, the latter turned out to be more disruptive as only about five games were released per week before Steam Greenlight which was an increase by a factor of 14 [McAloon 2018]. With the release of Steam Direct the documentation of it as well as Steamworks and its SDK was reworked and is now publicly available.

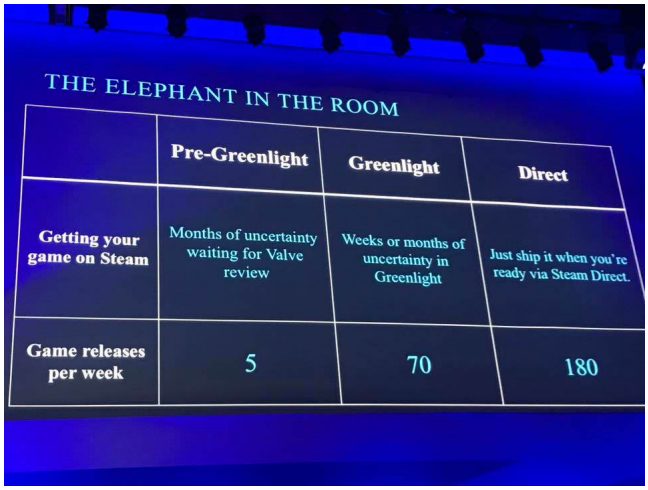

Figure 33: Weekly game releases with Steam Direct [McAloon 2018]

It should also be noted that most of these released games are indie titles, but the time Steam users spend playing indie titles is not keeping up, as visualized in figure 34, making creating successful indie games increasingly challenging.

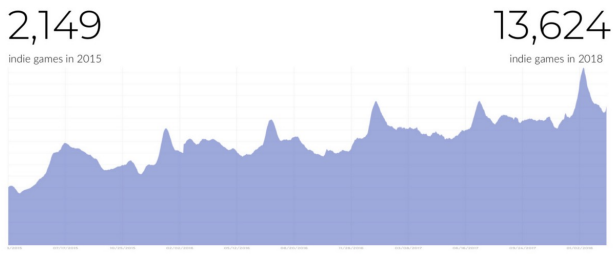

Figure 34: Total playtime of indie games on Steam compared to number of released titles [Galyonkin 2018]

5.4 Steamworks SDK

The Steamworks development SDK allows game developers to implement features expected by players on Steam, such as the Steam overlay, which is used to keep interacting with Steam, chat, browse the store and see leaderboard rankings and achievements being unlocked [Valve 2018l].

The SDK supports C++ Visual Studio 2008 or higher on Windows and GCC 4.6 or higher as well as Clang 3.0 or higher on macOS and Linux. It was used by Swarmlake to add support for the Steam overlay, achievements, and leaderboards.

5.4.1 Overlay

The initialization of Steam within the game and its overlay is done through the SteamAPI_Init() function. Afterwards, games can use a license check to see if the user is authenticated on the platform and owns the game by calling the SteamAPI_RestartAppIfNecessary() function which alternatively can also be added by an external DRM wrapper.

After the game has successfully launched, most code interactions happen through asynchronous events due to requiring networking to fetch or send information from or to the Steam servers. To receive callbacks to these events the game should bind custom functions to them when initiating an asynchronous query.

Those functions are then being activated if a result was received by calling the SteamAPI_RunCallbacks() function with a recommended interval of at least ten times per second. Finally, when closing the game, the resources should be released by calling the SteamAPI_Shutdown() function [Valve 2018l].

5.4.2 Achievements

Achievements provide additional value for players by ideally giving them optional and challenging tasks to increase the playtime. The configuration of their ids, images, names, and descriptions is being done through the Steamworks App Admin backend, as shown in figure 35.

To be able to fetch and cache the state of all achievements from the server, the game needs to call the RequestCurrentStats() function and bind to the OnUserStatsReceived() callback to receive the requested response. Afterwards, each achievement can be further queried for their unlocked state locally by using the GetAchievement() method.

On the other hand, unlocking achievements is done through the SetAchievement() function followed by calling StoreStats() with the respective OnUserStatsStored() callback to send the request to the server [Valve 2018m].

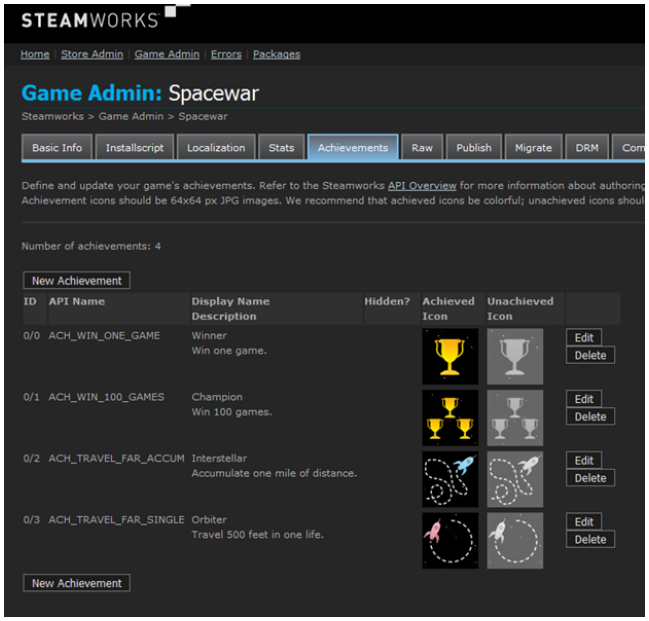

Figure 35: Steam achievements configuration on Steam App Admin backend [Valve 2018m]

Achievements in Swarmlake were used to give multiple subgoals towards reaching the final goal of getting a score of 10,000, which can be seen in figure 36. Additionally, each achievement contains an image of the alphabet to allow users to spell words on their achievements showcase within their game profile.

The method of using desirable achievements was also employed in an extreme form by many so-called fake games, which awarded players with thousands of achievements in some cases by merely idling in the game to quickly increase the user's achievement count.

Figure 36: Swarmlake achievements publicly visible on the community page

Due to Valve's continued effort, those fake games were later stopped from being easily profitable. That was done by employing a limit of 100 achievements, all achievements will no longer contribute towards the player's global achievement count and achievements cannot be shown on the profile until the game reaches a specific undisclosed confidence metric [Valve 2018d].

5.4.3 Leaderboard

Similar to achievements, leaderboards provide more value to players by allowing them to compete against friends or each other globally in games that have a scoring mechanic.

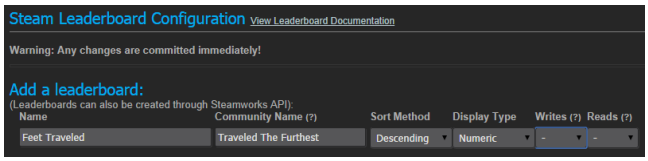

Figure 37: Steam leaderboards configuration on Steam App Admin backend [Valve 2018i]

The configuration on the Steamworks App Admin backend is about setting up a leaderboard with an id, a name, if it should be visible on the Steam community page as well as primary display and sort settings, as is shown in figure 37.

First, the leaderboard handle needs to be received from the server to be able to further work with a leaderboard in the game. This is done through the FindLeaderboard() function and the respective OnLeaderboardFound() callback.

Afterwards, leaderboard entries can be downloaded and cached with the DownloadLeaderboardEntries() method by providing the handle and the result will be returned using the OnLeaderboardScoresDownloaded() callback.

To further parse and get the state of those downloaded entries, the GetDownloadedLeaderboardEntry() function is used with the parameter of the entry index.

Finally, to upload a new or updated entry, the game needs to call the UploadLeaderboardScore() method by providing the leaderboard handle and the upload result can be checked in the OnLeaderboardScoreUploaded() callback [Valve 2018i].

In Swarmlake, a single leaderboard is used, as seen in figure 38, which is also being shown in the game to motivate players to improve their highscore. Additionally, it allows the player to measure their own skill at the game compared to others and provide an additional challenge after reaching the final required achievement score of 10,000.

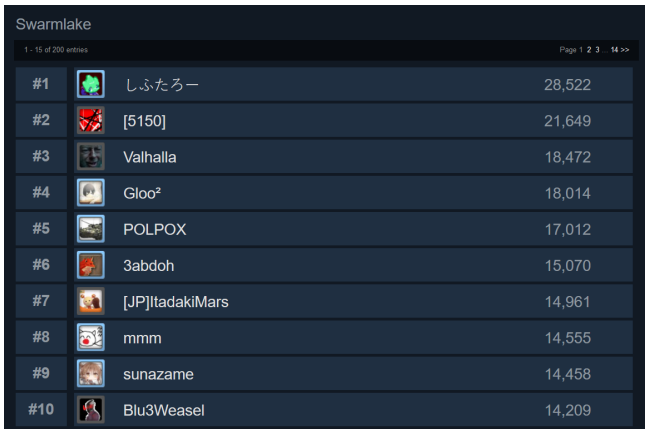

Figure 38: Swarmlake leaderboard publicly visible on the community page

It should be noted that there is little to no security from this Steam service to make sure that the scores are authentic. Management of the leaderboard is also limited with the only option, to edit or delete single entries or reset the whole leaderboard.

Since players cannot be banned from a leaderboard, different workarounds exist such as creating a new manually managed leaderboard which contains all banned players that is checked by the game initially. However, it might also be possible for attackers to upload scores even without using the game, which would circumvent that method.

Furthermore, cheating is hard to prevent in an offline single-player game since attackers can add a debugger or otherwise hack the game to do what they want. One possible strategy is to allow user verification by uploading a watchable replay along with the score if the game supports that.

Due to technological limitations, in Swarmlake only a simple non-disclosed solution was used to prevent the easiest of memory hacks from working with standard tools such as Cheat Engine without further knowledge of the game's internal handling.

5.4.4 Cloud Saves

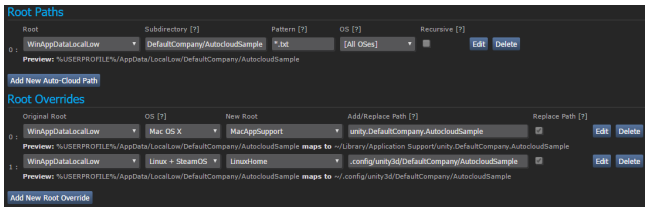

Figure 39: Steam auto-cloud configuration on Steam App Admin backend [Valve 2018f]

Cloud saves allow to store saved game data such as progress on the Steam servers so that players will be able to keep their save data across different hardware or when removing the local storage.

The configuration of this service is done in the Steamworks App Admin backend where the developer can set the byte quota per user and the number of files allowed, as displayed in figure 39.

Steam cloud can be integrated using the Steamworks API or by using auto-cloud, which tells the Steam client to synchronize the

specified files after termination automatically.

That is done based on the configuration of a base directory path and file patterns with optional wildcards which should exclude machine specific configuration like video settings [Valve 2018f].

5.4.5 Trading Cards

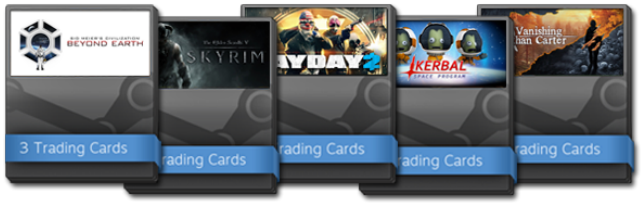

Figure 40: Steam Trading Card Booster Packs [Valve 2018k]

Steam Trading Cards, as seen in figure 40, were added in 2013 so that players can collect virtual cards to show off their favorite games. They are tradable with other players using real money, which allowed them to create their own ecosystem [Valve 2013].

Steam Trading Cards are being awarded automatically after a certain amount of playtime after the developer has set them up in the Steamworks App Admin backend and once they are approved by Valve [Valve 2018k].

They include the following:

- Five to 15 trading cards in 1920x1080 and 206x184
- Five to ten emoticons in 18x18 and 54x54
- Five badges in 80x80
- One foil badge in 80x80
- Three to ten profile backgrounds in 1920x1080

Trading cards add extra value for users as well as developers which implement them, but similar to achievements those virtual cards also attracted abuse by fake games.

Therefore, Steam Trading Cards are now only approved by Valve after reaching the non-disclosed confidence metric related to the performance of how well the game is doing [Valve 2017a].

5.5 Marketing

It was estimated that Steam made a total of about $4.3B revenue with approximately 291M active users in 2017 [Bailey 2018b]. Valve's digital game distribution service is influential for the PC gaming market and accounts for at least 18% revenue of this gaming platform, which excludes all in-game purchases such as DLC (downloadable content) [Bailey 2018a].

The games sold on Steam are hit-driven, since 50% of all earned revenue of the platform came from the top 100 (0.5%) titles. Those games require about $22M earned yearly revenue to get into the top 20 which has been relatively unchanged from 2016.

The median owners for indie games, on the other hand, went down from approximately 5,000 overall to about 1,500 for games released in 2017. Similarly, median prices of indie games have decreased from $3.99 overall to about $2.99 for released games in 2017.

Most of this could be due to the increased amount of games that were released with the Steam Direct launch in the middle of 2017. That could be inferred due to the marginally higher gross revenue of $160,000 that was required in 2017 to reach the top 2,000 games compared to the $150,000 gross revenue the year before [Galyonkin 2018].

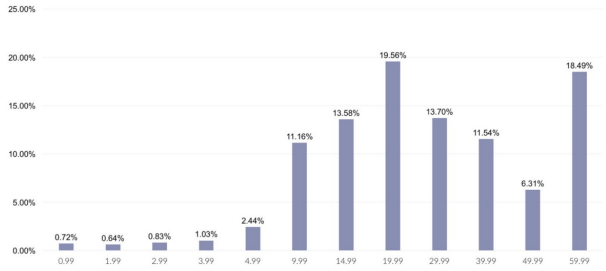

Figure 41: Revenue by max price excluding the outlier game PlayerUnknown's Battlegrounds [Galyonkin 2018]

The approximate normal distribution of Steam game prices is around $9.99, as shown in figure 41, where many indie games are located. However, the amount of revenue by titles earned at that price range is only 11.16% compared to $19.99 with about 19.56% and $59.99 with 18.49%.

Swarmlake was sold at $0.99, whose price category is responsible for about 0.72% global revenue and later at $2.99, whose price range translates to about 0.83% of all revenue for Steam games. That also shows that cheap games do not sell as well as people are gravitating towards more expensive titles with a possibly higher perceived quality.

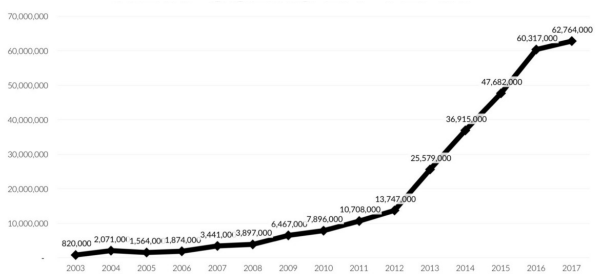

Figure 42: New users on Steam per year [Galyonkin 2018]

Steam has a steadily growing user base with more and more new players joining every year, which is displayed in figure 42. About 63,000 new users have joined in 2017, which equals to approximately 22% of all players.

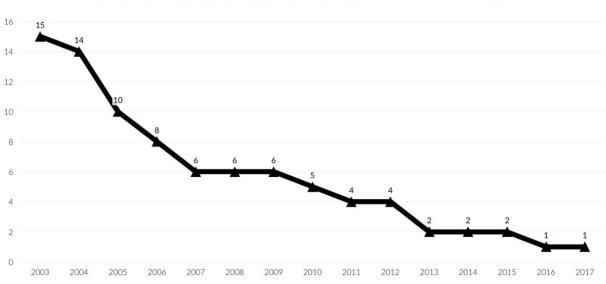

Figure 43: Median games per users owned on Steam [Galyonkin 2018]

However, new players are buying fewer games, which results in a shrinking count of median games per user owned, as shown in figure 43.

The top 10 countries buying the most amount of indie games are the following:

1. USA
2. China
3. Russia
4. Germany
5. UK
6. France
7. Canada
8. Brazil
9. Poland
10. Australia

That could mean that spending time or money into localizing the game might be worthwhile to find the biggest audience. On the other hand, the top 10 countries spending the most amount of money on indie games are different due to the special regional pricing of markets like China and Russia:

1. USA
2. Germany
3. UK
4. France
5. Canada
6. Australia
7. Poland
8. Russia
9. China
10. Brazil

Furthermore, Steam also features discounts displayed on the store, which helps boost the visibility of participating titles. The platform has at least two big yearly sales as well, which are the Steam Winter Sale and Steam Summer Sale.

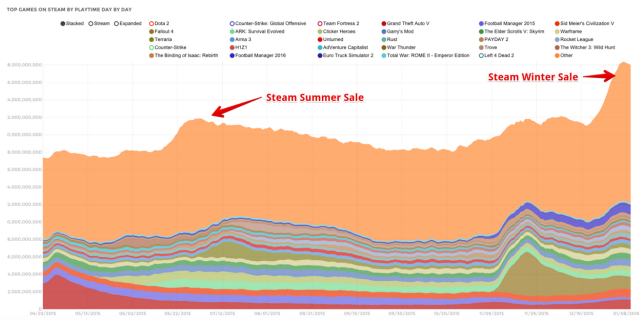

Figure 44: Playtime of Steam games [Galyonkin 2016]

The former is usually more prominent than the latter and made about $270M in the winter of 2016, which is twice as much as the sale in the summer of the same year [Galyonkin 2016]. The reason for this could be that people have more time to play during the winter sale, as can be seen in figure 44.

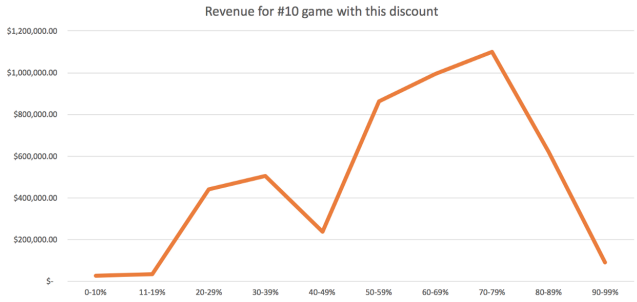

Figure 45: Steam game sales by discount [Galyonkin 2016]

As of 2017, indie games sell in average 21,000 copies over their lifetime with the average price of $8.72, which was discounted in average to about $4.63 during the Steam Summer Sale 2016 [Galyonkin 2017]. Additionally, out of all usual discounts, the most lucrative are at 75%, 66%, and 50%, as displayed in figure 45.

Discounting games on Steam is an often used and great tool to grow a title's audience and increase its lifetime. The following types of discounts exist on the platform:

- Launch discount: Lasts for seven days and can be up to 40% off of the normal price
- Weeklong Deal: Lasts for seven days starting on Monday 10 AM Pacific time where any title can take part and is featured in a capsule on the store
- Custom discount: Can be fully customized
- Daily Deal: Lasts for 48 hours and is curated by Valve
- Weekend Deal: Lasts for four days starting on Thursday 10 AM Pacific time and is curated by Valve
- Midweek Madness: Lasts for three days starting on Tuesday 10 AM Pacific time and is curated by Valve
- Seasonal sales: Special holiday sales such as Winter Sale or Summer Sale where any game can take part and which are generating much activity on the platform

Self-serve discounts can be created and managed in the Steamworks page, and for curated promotions, Valve will contact the developers or publishers themselves. Additionally, the following rules exist to prevent developers or publishers from creating discounts too often and protecting the value of their games:

1. Custom discounts need to be spaced apart by at least two months
2. Custom discounts can only be submitted after two months of release
3. Seasonal discounts can only be joined after 30 days of release or 30 days of the end of the launch discount

While discounting is a powerful tool, it also needs to be carefully handled not to ruin longterm sales and revenue. Valve suggests to stairstep and ease into the discount percentage over time by first reducing the price by 33%, which is followed by 50%, 66% and 75% over the course of a year.

Additionally, scheduling content updates of the game to discounts can further increase player interest as it sends the message that the developer is committed to improving the game at a the time where the title has increased visibility [Valve 2018h]. For Swarmlake, participating in the Steam holiday sales as well as weeklong sales has been essential to raising the awareness and discoverability of the game.

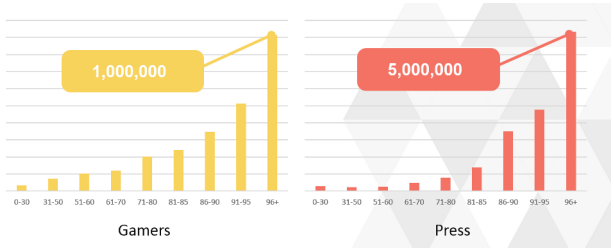

Figure 46: Press game review scores and player count correlation [Galyonkin 2015]

Good reviews from the press, as well as user reviews, usually correlate with good sales, as displayed in figure 46. Nonetheless, bad reviews are likely still better than silence, which emphasizes the need to invest in PR and marketing.

This is further demonstrated by successful games on Steam usually having press coverage for a sizable time before release. To stand out from the crowd, it is a useful strategy to try to find the right niche for the game [Galyonkin 2015].

Therefore, games should also be treated as a unique marketing problem to create the best possible PR strategy for. It is also unlikely for titles to be the best they can be on launch day so that day should instead be treated as the starting line instead of the finishing line, with additional community feedback shaping the final versions [Valve 2018j].

6 Conclusion

Video games are complex real-time applications which require many iterations during development and need to be performant on release. Understanding and choosing the correct game loop for the title is an essential first task when designing the game engine.

Afterwards, when thinking about and architecting for data locality, it is possible to create a game that can handle a massive amount of objects on the CPU. Instanced rendering has a similar goal to handle rendering those amounts of objects on the GPU by reducing draw calls.

Most games also require creating and destroying game objects on the fly, which can be efficiently implemented using object pooling. Designing a multi-threaded game engine with data decomposition can benefit interactive software today and in the future by being able to handle different hardware.

Expensive tasks such as collision detection, which requires about 4.4ms (68%) of approximately 6.5ms frame-time in Swarmlake, can also be improved using spatial partitioning. More low-level CPU optimization strategies exist and should be done at the end of the development cycle, such as using the ”fast floating-point operations” compiler option which resulted in about a 1ms frame-time speedup.

Sorting objects and object batches by their distance to effectively use early depth testing of modern GPUs can also speed up rendering. For enemies in Swarmlake, where distance cannot be predicted, sorting the batches by the used model vertex count resulted in a frame-time performance speedup of about 0.7ms.

Displaying only objects within the view frustum of the camera can result in a speedup, which in Swarmlake reduced the total render-time by about 1.2ms in average cases. It is also suggested to optimize shaders and approximate effects such as using downsampling to get a wide bloom, which reduced the total render-time by about 2ms in the game.

Finally, selling a product such as a video game requires marketing and distribution, with the latter usually being done through Steam if it is a PC game. Integrating the SDK into the game, as well as understanding the history of Steam including Steam Greenlight and current systems such as Steam Direct and sales can be beneficial to developing and marketing the title on the platform correctly.

6.1 Successes

Due to the optimization strategies explained in this article, it was possible to create a unique experience of fighting more than 10,000 enemies at once in Swarmlake.

Additionally, the choice of using a low-poly art style as well as cheap and fast post processing effects such as HDR, tonemapping, bloom, vignette, chromatic aberration, grain and anti-aliasing allowed for a modern visual presentation, as seen in figure 47 and figure 48. This was done to attract potential customers while handling the needed amount of objects.

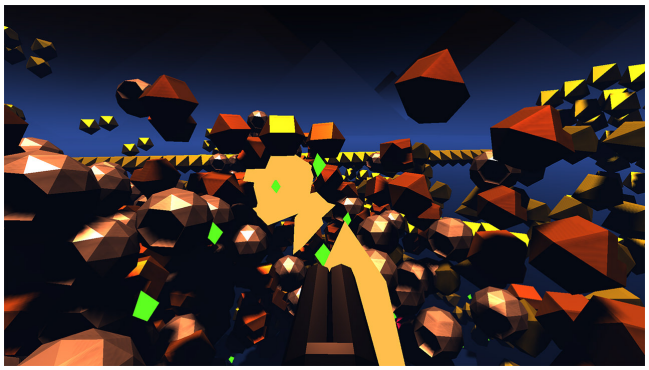

Figure 47: Swarmlake first color pass render without any post processing effects

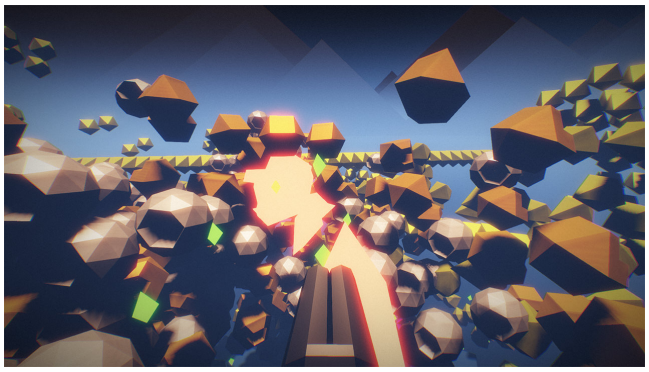

Figure 48: Swarmlake final pass render with all post processing effects

Swarmlake has received continued updates for at least half a year after release based on player feedback, which allowed the game to get increasingly better and reach the overwhelmingly positive user review label on Steam. This means that the title has gotten more than 500 user reviews of which at least 95% are positive, which should help the game to gain further visibility and awareness in the long run.

Swarmlake was developed over the course of two and a half years almost exclusively by a single developer, with $550 extra cost for the soundtrack, font and sound effects. As of September 30, 2018, the game sold 21,757 units and grossed at $14,192 in revenue, which still includes VAT (value-added tax), taxes, refunds and the revenue cut for Steam.

6.2 Pitfalls

It was clear from the beginning that Swarmlake could not be created as intended without careful optimization. Therefore the development team tried to optimize as early and as much as possible, which made it difficult to iterate and make the game fun and also very time intensive to finish the project.

The unique selling point of Swarmlake also might have been incorrectly communicated based on initial player feedback and the low amount of sales despite overwhelmingly positive reviews. In addition to that and due to no time or money being available to market the game it was released at the lowest possible price of $0.99 on Steam, which could have further resulted in a low perceived value.

Half a year after the release of Swarmlake, the overwhelmingly positive user rating label was received. Since the game was more polished at that point, the price was increased to the intended amount of $2.99, which however resulted in currently noticeably reduced revenue and fewer units sold, as shown in figure 49.

It is unclear if this is just a short-term effect, but it is suggested to sell a game at the intended price point from the start [Galyonkin 2015].

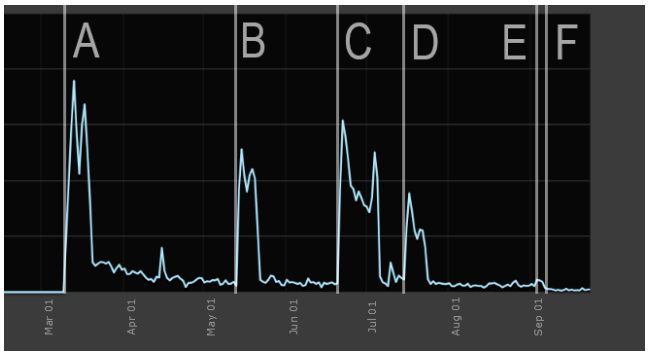

Figure 49: Relative Swarmlake units sold. A: Release at $0.99 and 20% discount, B: Weeklong sale at 40% discount, C: Steam Summer Sale at 50% discount, D: Weeklong sale at 50% discount, E: Overwhelmingly positive user review label received, F: Price increased to $2.99

References

AKENINE-MOLLER, T., HAINES, E., AND HOFFMAN, N. 2008. *Real-Time Rendering*, 3rd ed. A. K. Peters, Ltd., Natick, MA, USA.

AKHTER, S., AND ROBERTS, J. 2006. *Multi-core programming*, vol. 33. Intel press Hillsboro.

ALBRECHT, T., 2009. Pitfalls of object oriented programming. https://github.com/Michaelangel007/game_dev_pdfs/blob/master/c++/Pitfalls_of_Object_Oriented_Programming_GCAP_09.pdf. [Accessed: 12-October-2018].

BAILEY, D., 2018. Pc game sales numbers market share 2017. https://www.pcgamesn.com/pc-game-sales-numbers-market-share-2017. [Accessed: 12-October-2018].

BAILEY, D., 2018. Steam revenue 2017. https://www.pcgamesn.com/steam-revenue-2017. [Accessed: 12-October-2018].

BINSTOCK, A., 2011. Data decomposition: Sharing the love and the data. https://software.intel.com/en-us/articles/data-decomposition-sharing-the-love-and-the-data. [Accessed: 12-October-2018].

CALVIN, 2018. Shadow mapping. http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/. [Accessed: 12-October-2018].

CATO, V., 2013. C++ fastest way to clear or erase a vector. https://stackoverflow.com/questions/16420357/c-fastest-way-to-clear-or-erase-a-vector#16420529. [Accessed: 12-October-2018].

CHIANG, O., 2011. The master of online mayhem. https://www.forbes.com/forbes/2011/0228/technology-gabe-newell-videogames-valve-online-mayhem.html#34d43b6d3ac0. [Accessed: 12-October-2018].

CHOPANZA, J., 2014. Want speed? don't (always) pass by value. https://juanchopanzacpp.wordpress.com/2014/05/11/want-speed-dont-always-pass-by-value/. [Accessed: 12-October-2018].

CLANG, 2018. Performance inefficient vector operation. http://releases.llvm.org/6.0.1/tools/clang/tools/extra/docs/clang-tidy/checks/performance-inefficient-vector-operation.html. [Accessed: 12-October-2018].

COORG, S., AND TELLER, S. 1997. Real-time occlusion culling for models with large occluders. In *Proceedings of the 1997 symposium on Interactive 3D graphics*, ACM, 83–ff.

CRIDER, M., 2017. What is coil whine, and can i get rid of it on my pc? https://www.howtogeek.com/297166/what-is-coil-whine-and-can-i-get-rid-of-it-on-my-pc/. [Accessed: 12-October-2018].

DE VRIES, J., 2018. Opengl instancing. https://learnopengl.com/Advanced-OpenGL/Instancing. [Accessed: 12-October-2018].

DELLAFAVE, R., 2014. Tips for getting greenlit on steam greenlight. https://gamedevelopment.tutsplus.com/articles/tips-for-getting-greenlit-on-steam-greenlight--gamedev-13938. [Accessed: 12-October-2018].

DRAXINGER, W., 2015. Opengl tearing with fullscreen native resolution. https://stackoverflow.com/questions/30293074/opengl-tearing-with-fullscreen-native-resolution#30300615. [Accessed: 12-October-2018].

DUNN, J., 2013. Full steam ahead: The history of valve. https://www.gamesradar.com/history-of-valve/. [Accessed: 12-October-2018].

EITZ, M., AND LIXU, G. 2007. Hierarchical spatial hashing for real-time collision detection. In *null*, IEEE, 61–70.

FABIAN, R. 2013. *Data-Oriented Design*. June.

FAN, Z., LI, H., HILLESLAND, K., AND SHENG, B. 2015. Simulation and rendering for millions of grass blades. In *Proceedings of the 19th symposium on interactive 3D graphics and games*, ACM, 55–60.

FERNANDES, R. M., 2011. What are aggregates and pods and how/why are they special? https://stackoverflow.com/questions/4178175/what-are-aggregates-and-pods-and-how-why-are-they-special#7189821. [Accessed: 12-October-2018].

FIEDLER, G., 2004. Fix your timestep! https://gafferongames.com/post/fix_your_timestep/. [Accessed: 12-October-2018].

GALYONKIN, S., 2015. Some things you should know about steam. https://galyonk.in/some-things-you-should-know-about-steam-5eaffcf33218. [Accessed: 12-October-2018].

GALYONKIN, S., 2016. About steam winter sale. https://galyonk.in/about-steam-winter-sale-76a75abe152a. [Accessed: 12-October-2018].

GALYONKIN, S., 2017. The indie games are too damn cheap. https://galyonk.in/the-indie-games-are-too-damn-cheap-11b8652fad16. [Accessed: 12-October-2018].

GALYONKIN, S., 2018. Steam in 2017. https://galyonk.in/steam-in-2017-129c0e6be260. [Accessed: 12-October-2018].

GORDON, R. C., 2014. Game development with sdl 2.0. http://media.steampowered.com/apps/steamdevdays/slides/sdl2.pdf. [Accessed: 12-October-2018].

HOGG, J., 2015. Do you prefer fast or precise? https://blogs.msdn.microsoft.com/vcblog/2015/10/19/do-you-prefer-fast-or-precise/. [Accessed: 12-October-2018].

JR, C., 2016. Spatial hashing in c++. http://www.sgh1.net/posts/spatial-hashing-1.md. [Accessed: 12-October-2018].

JUKIĆ, T., 2015. Draw calls in a nutshell. https://medium.com/@toncijukic/draw-calls-in-a-nutshell-597330a85381. [Accessed: 12-October-2018].

KHRONOS, 2014. Glsl step. https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/step.xhtml. [Accessed: 12-October-2018].

KHRONOS, 2018. Depth test. https://www.khronos.org/opengl/wiki/Depth_Test. [Accessed: 12-October-2018].

KHRONOS, 2018. Swap interval. https://www.khronos.org/opengl/wiki/Swap_Interval. [Accessed: 12-October-2018].

LEFEBVRE, S., AND HOPPE, H. 2006. Perfect spatial hashing. In *ACM Transactions on Graphics (TOG)*, vol. 25, ACM, 579–588.

LIGHTHOUSE3D, 2011. View frustum culling. http://www.lighthouse3d.com/tutorials/view-frustum-culling/. [Accessed: 12-October-2018].

LLOPIS, N. 2009. Data-oriented design (or why you might be shooting yourself in the foot with oop). *Game Developer* (September), 43–45.

MARSHALL-NAGY, D., 2014. Case study: The dynamic history of valve. from game developer to console manufacturer. http://www.adaptivecycle.nl/images/Mini_case_study_Valve.pdf. [Accessed: 12-October-2018].

MCALOON, A., 2018. Steam direct sees 180 game releases per week, over twice as many as greenlight did. https://www.giantbomb.com/steam-greenlight/3015-7583/. [Accessed: 12-October-2018].

MEIRI, E., 2018. Instanced rendering. http://ogldev.atspace.co.uk/www/tutorial33/tutorial33.html. [Accessed: 12-October-2018].

MEYERS, S., 2013. Cpu caches and why you care.

MITTRING, M., 2012. The technology behind the unreal engine 4 elemental demo.

NYSTROM, B. 2014. *Game Programming Patterns*. Genever Benning, November.

ORION GRANATIR, O. R., 2010. Don't dread threads. https://slideplayer.com/slide/6665648/. [Accessed: 12-October-2018].

PAUL, B., 1997. Opengl performance optimization. http://www.inf.pucrs.br/flash/tcg/aulas/opt/opengl_perf_opt.html#Clearing. [Accessed: 12-October-2018].

PEARSON, J., 2017. The first text adventure game ever is finally open source. https://motherboard.vice.com/en_us/article/ywmyn5/the-first-text-adventure-game-ever-is-finally-open-source. [Accessed: 12-October-2018].

PERSSON, E., 2013. Low-level thinking in high-level shading languages. http://www.humus.name/Articles/Persson_LowLevelThinking.pdf. [Accessed: 12-October-2018].

PERSSON, E., 2014. Low-level shader optimization for next-gen and dx11. http://www.humus.name/Articles/Persson_LowlevelShaderOptimization.pdf. [Accessed: 12-October-2018].

PLACZEK, M., 2016. Object pooling in unity. https://www.raywenderlich.com/847-object-pooling-in-unity. [Accessed: 12-October-2018].

POZO, R. 1997. Template numerical toolkit for linear algebra: High performance programming with c++ and the standard template library. *The International Journal of Supercomputer Applications and High Performance Computing 11*, 3, 251–263.

SAYER, M., 2016. The 13-year evolution of steam. https://www.pcgamer.com/steam-versions/. [Accessed: 12-October-2018].

SHARP, J. A. 1980. Data oriented program design. *ACM SIGPLAN Notices 15*, 9, 44–57.

SHREINER, D., GROUP, B. T. K. O. A. W., ET AL. 2009. *OpenGL programming guide: the official guide to learning OpenGL, versions 3.0 and 3.1*. Pearson Education.

STOLK, B., 2016. Fixing your time step, the easy way with the golden 4.16 ms. https://www.gamasutra.com/blogs/BramStolk/20160408/269988/Fixing_your_time_step_the_easy_way_with_the_golden_48537_ms.php. [Accessed: 12-October-2018].

STUDIOCOAST, 2018. Screen tearing. https://www.vmix.com/knowledgebase/article.aspx/46/screen-tearing. [Accessed: 12-October-2018].

TIAN, X., SAITO, H., GIRKAR, M., PREIS, S., KOZHUKHOV, S., CHERKASOV, A. G., NELSON, C., PANCHENKO, N., AND GEVA, R. 2012. Compiling c/c++ simd extensions for function and loop vectorizaion on multicore-simd processors. *IEEE 26th International Parallel and Distributed Processing Symposium Workshops and PhD Forum*, 2349–2358.

VALVE, 2012. First titles get steam's greenlight. https://store.steampowered.com/news/8839/. [Accessed: 12-October-2018].

VALVE, 2012. Valve launches steam greenlight. https://store.steampowered.com/news/8761/. [Accessed: 12-October-2018].

VALVE, 2012. What we're doing about discoverability in steam greenlight. https://steamcommunity.com/games/765/announcements/detail/1317556891741839763. [Accessed: 12-October-2018].

VALVE, 2013. Steam trading cards released. https://store.steampowered.com/news/10946/. [Accessed: 12-October-2018].

VALVE, 2017. Changes to trading cards. https://steamcommunity.com/games/593110/announcements/detail/1954971077935370845. [Accessed: 12-October-2018].

VALVE, 2017. Closing greenlight, steam direct launches. https://steamcommunity.com/games/593110/announcements/detail/1265922321514182595. [Accessed: 12-October-2018].

VALVE, 2017. Evolving steam. https://steamcommunity.com/games/593110/announcements/detail/558846854614253751. [Accessed: 12-October-2018].

VALVE, 2017. Steam client update released. https://store.steampowered.com/news/35534/. [Accessed: 12-October-2018].

VALVE, 2018. Controller gaming on pc. https://steamcommunity.com/games/593110/announcements/detail/1712946892833213377. [Accessed: 12-October-2018].

VALVE, 2018. Introducing a new version of steam play. https://steamcommunity.com/games/221410/announcements/detail/1696055855739350561. [Accessed: 12-October-2018].

VALVE, 2018. Joining the steamworks distribution program. https://partner.steamgames.com/steamdirect. [Accessed: 12-October-2018].

VALVE, 2018. More changes addressing fake games. https://steamcommunity.com/groups/steamworks#announcements/detail/3077529424431732424. [Accessed: 12-October-2018].

VALVE, 2018. Progress update. https://steamcommunity.com/games/593110/announcements/detail/1708442022337025126. [Accessed: 12-October-2018].

VALVE, 2018. Steam cloud. https://partner.steamgames.com/doc/features/cloud. [Accessed: 12-October-2018].

VALVE, 2018. Steam direct now available. https://steamcommunity.com/games/593110/announcements/detail/1328973169870947116. [Accessed: 12-October-2018].

VALVE, 2018. Steam discounting. https://partner.steamgames.com/doc/marketing/discounts. [Accessed: 12-October-2018].

VALVE, 2018. Steam leaderboards. https://partner.steamgames.com/doc/features/leaderboards. [Accessed: 12-October-2018].

VALVE, 2018. Steam marketing best practices. https://partner.steamgames.com/doc/marketing/bestpractices. [Accessed: 12-October-2018].

VALVE, 2018. Steam trading cards. https://partner.steamgames.com/doc/marketing/tradingcards. [Accessed: 12-October-2018].

VALVE, 2018. Steamworks api overview. https://partner.steamgames.com/doc/sdk/api. [Accessed: 12-October-2018].

VALVE, 2018. Step by step: Achievements. https://partner.steamgames.com/doc/features/achievements/ach_guide. [Accessed: 12-October-2018].

VALVE, 2018. Who gets to be on the steam store? https://steamcommunity.com/games/593110/announcements/detail/1666776116200553082. [Accessed: 12-October-2018].

WICHT, B., 2012. C++ benchmark – std::vector vs std::list vs std::deque. https://baptiste-wicht.com/posts/2012/12/cpp-benchmark-vector-list-deque.html. [Accessed: 12-October-2018].

WILDE, T., 2018. What happened to steam machines? https://www.pcgamer.com/what-happened-to-steam-machines/. [Accessed: 12-October-2018].

WINGFIELD, N., 2012. Game maker without a rule book. https://www.nytimes.com/2012/09/09/technology/valve-a-video-game-maker-with-few-rules.html. [Accessed: 12-October-2018].

WITTERS, K., 2009. dewitters game loop. http://www.koonsolo.com/news/dewitters-gameloop/. [Accessed: 12-October-2018].

WOLFE, A., 2015. Avoiding the performance hazzards of std::function. https://blog.demofox.org/2015/02/25/avoiding-the-performance-hazzards-of-stdfunction/. [Accessed: 12-October-2018].

WRIGHT JR, R. S., HAEMEL, N., SELLERS, G. M., AND LIPCHAK, B. 2010. *OpenGL SuperBible: comprehensive tutorial and reference*. Pearson Education.